

WSDL-Based Automatic Test Case Generation for Web Services Testing

Xiaoying Bai, Wenli Dong

*Department of Computer Science and Engineering, Tsinghua University
Beijing, China, 100084*

baixy@mail.tsinghua.edu.cn, wldong@mail.tsinghua.edu.cn

Wei-Tek Tsai, Yinong Chen

*Computer Science & Engineering Department, Arizona State University
Tempe, AZ 85287-8809, U.S.A*

Wei-tek.tsai@asu.edu, yinong@asu.edu

Abstract

Web Services promote the specification-based cooperation and collaboration among distributed applications in an open environment. To ensure the quality of the services that are published, bound, invoked and integrated at runtime, test cases have to be automatically generated and testing executed, monitored and analyzed at runtime. This paper presents the research to generate Web Services test cases automatically based on the Web Services specification language WSDL (Web Services Description Language), which carries the basic information of a service including its interface operations and the data transmitted. The WSDL file is first parsed and transformed into the structured DOM tree. Then, test cases are generated from two perspectives: test data generation and test operation generation. Test data are generated by analyzing the message data types according to standard XML schema syntax. Operation flows are generated based on the operation dependency analysis. Three types of dependencies are defined: input dependency, output dependency, and input/output dependency. Finally, the generated test cases are documented in XML-based test files called Service Test Specification.

1. Introduction

Web Services (WS) is the enabling technique for Service-Oriented Computing (SOC), which provides a standard-based mechanism and open platform for integrating distributed autonomous service components [12]. The quality of services is a key issue for developing service-based software systems, and testing is necessary for evaluating the functional correctness,

performance and reliability of individual as well as composite services [1][2].

However, the dynamic features of WS impose numerous new challenges to traditional testing techniques. First, services are published, bound, invoked and integrated at runtime. To incorporate into the framework, testing needs to be automated including automatic test case generation, test execution, test results collection and analysis. Second, WS proposes a fully specification-based process using a set of XML-based standards, e.g. SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language)[12][13], UDDI (Universal Description, Discovery and Integration) and WSFL (Web Services Flow Language). Rather than graphical user interfaces, services are provided through programmable interfaces, which are invisible to end users. Hence, test cases must be generated based on the standard specifications and test tools need to analyze the specifications to extract necessary information such as interface operations, data structures, and operation semantics. Third, to use and reuse test cases in the open environment and through service evolutions, test cases should be documented following the XML-based standard format.

Up-to-now, most of the research of WS testing is still theoretically based on formal methods such as model checking. For examples, Shin proposes a model-checking technique to verify service flows using automation-based model checker SPIN [5]. Howard et. al propose FSP (Finite State Process) notation to model and verify service composition [3]. X. Yi and K. J. Kochut propose a CP-nets model, an extended Petri Net model, for specifying and verifying web service composition [10]. Srini and Sheila adopt DAML-S ontology to describe web service semantic and try to test web services by simulating its execution under

different input conditions [6]. Tsai et al proposed to generate test cases based on OWL-S specification during the completeness & consistency analysis and model checking [7][8][9]. WS testing tools also begin to emerge in recent years [14][15][16][17]. These tools facilitate monitoring and debugging of WS execution, such as trace the process, debug WSDL and SOAP messages, capture and replay the requests and responses, and check the syntax and semantic correctness of the specifications.

This paper proposes a technique for generating test cases automatically based on WSDL specification. The WSDL-based test case generation is a part of the distributed service testing framework that includes test case generation, test controller, test agents and test evaluator. The specification of the service is first parsed into a DOM tree representation based on the WSDL schema. Test cases are generated from four levels: test data generation, individual test operation generation, operation flow generation, and test specification generation. Test data are generated by analyzing the messages based on XML schema data type definition including simple data type, complex

data type, aggregated data type and user-derived data type. Tests for individual operations are generated by analyzing the associated parameters (i.e. the messages). A service may contain multiple operations. To test the combinational functions, tests are generated as sequence of operations. The sequences are generated based on operation dependence analysis including input dependency, output dependency and input/output dependency. The generated test cases are recorded into a XML-based document call Service Test Specification (STS). STS takes the basic elements from WSDL including operations, input, output, message and part.

The rest of this paper is organized as follows. Section 2 introduces the overall architecture of distributed WS testing framework. Section 3 depicts the mechanism of WSDL-based test generation, including test data generation, operation flow generation and test specification. Section 4 discusses a case study. Section 5 summaries the paper and draws the conclusions.

2. The overall architecture

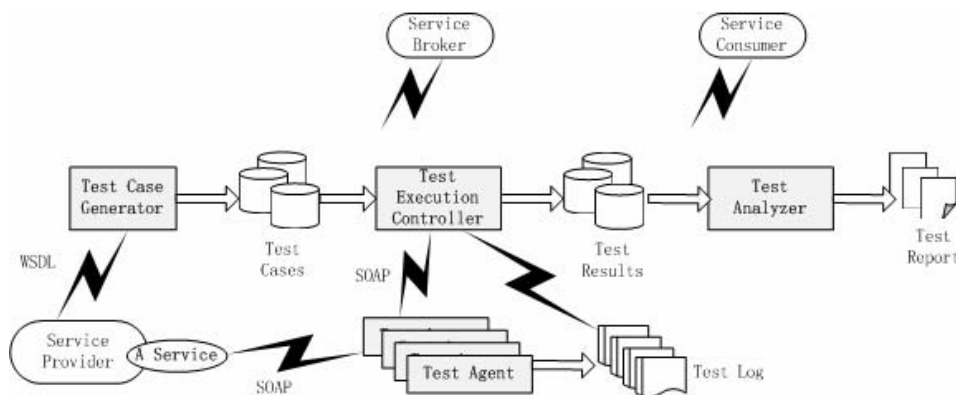


Figure 1 Overall architecture of the distributed service testing framework

Figure 1 shows the overall architecture of web service testing framework:

1. Test Case Generator accepts the service submission in WSDL and automatic generate test cases. The generated tests will be stored in the central test database. The generator can be extended to generate test cases based on other WS specification languages such as BPEL4WS and OWL-S.
2. Test Execution Controller controls test execution in a distributed environment. It retrieves test cases from the test database, allocates them to distributed test agents, monitors test runs, and collects test results.
3. Test Agents are dispersed in a LAN or WAN area.

An agent is a proxy that performs remote testing on target services with specific usage profiles and test data.

4. Test analyzer analyzes test results, evaluate the quality of services and produce test reports.

All the parties including service provider, service broker and service consumer can access the databases with different privileges.

This general framework can be applied to test services at different levels, such as:

- L1: Test the individual operations of an atomic service;
- L2: Test the combination operations of atomic services;
- L3: Test individual operations of composite

- services; and
- L4: Test the combination operations of composite services.

This paper focuses on testing the atomic services, that is, L1 and L2 testing. It presents the research of automatic test generation based on the specification of a service interface and the WSDL specification.

3. WSDL-Based test case generation

Based on the WSDL schema, test cases are generated from four levels as shown in Figure 2:

1. Test data are generated from WSDL message definition based on XML schema data types. Four types of data are discussed including simple data type, complex data type, aggregated data type and

user-derived data type.

2. Test operations are generated based on analysis of its associated parameters (messages).
3. Operation flows are generated to test a sequence of operations by operation dependency analysis. Three types of dependencies are identified: input dependency, output dependency and input/output dependency. The paper discusses how these dependencies can be used in test generation.
4. Test specification generation is to finally encode the test cases in XML files. Going through the levels, multiple test cases can be generated to test the service including individual operations as well as composite operations from both positive and negative perspectives.

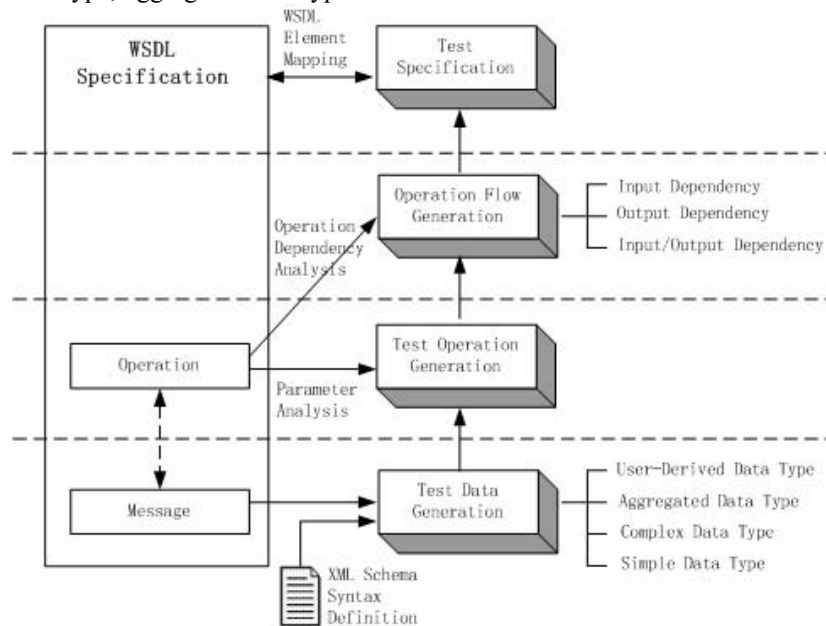


Figure2 WSDL-Based Test Case Generation

3.1. Test data generation based on XML Schema data type

According to the service description, messages are extracted, bound to the operations and analyzed to generate test cases. Test cases are generated based on data type analysis. The XML schema syntax defines two message types: simple type and complex type [11]. Different types can be combined to aggregated data types. Users can also define their own types derived from the build-in data types.

The XML standard defines a set of build-in simple data types, including primitive types and derived types. Each simple type is associated with a set of facets which characterize the particular aspects of a data type. Facets are of two types: fundamental facets

semantically abstract the data type, such as equal, ordered, bounded, cardinality and numeric; and constraining or non-fundamental facets are used to restrict the data values, such as length, pattern, enumeration, etc..

This paper discusses how to automatic generate test data based on data types and their associated constraining facets. A knowledge base is established where each build-in simple data type is associated with default facets definition and sets of candidate values based on the test strategies such as random values and boundary values. That is, SDT (Simple Data Type) is defined as <FACET, DV> where FACET is the set of named constraints, {<name_i, constraints_i>}; DV is the set of default values corresponding to different testing

strategies, $\{ \langle \text{strategy}_i, \{ \text{value}_k \} \rangle \}$. For example, type “string” is recorded with its facets and default values as $\{ \text{length}=\text{ANY}, \text{minLength}=0, \text{maxLength}=\text{MAX}, \text{patter}=\text{ANY}, \text{enumeration}=\text{ANY}, \text{whiteSpace}=\text{ANY}, \text{ordered}=\text{false}, \text{bounded}=\text{false}, \text{cardinality}=\text{countably infinite}, \text{numeric}=\text{false} \}$, where ANY is a pre-defined value meaning no restrictions. The type is also assigned a default set of random values $\{ \text{“abce”}, \text{“= +/~”}, \dots \}$. Once the WSDL file is parsed, the analyzer will extract the data, create instances of their corresponding data type, obtain and record the facet values with the data instances. Each facet is linked to a facet generator, for examples, for a data of type “string”, `lenGen ()` to generate the length of a string, `pattenGen()` to generate a string according to the patterns. Test generator coordinates the facets generators and composes the result data. The basic algorithm is as follows:

1. Retrieve the database for data type definition.
2. For each facet of the data type, if it is not defined in the instance, take its default value from the database; otherwise, take the instance value.
3. Invoke the facet generators in sequence to generate test data based on the facet definition of the instance.

Complex data type defines a composition of simple and/or complex data types. It defines the “sequence”, “choice”, and “all” relationship among member data types. To generate test data of complex data types, the generator recursively analyzes the structure of the data type until it reaches the simple type. The basic algorithm is as follows:

1. Analyze the hierarchical tree structure of a complex data type.
2. Traverse the tree, and at each tree node:
 - a) If it is a simple data type, perform simple data type analysis.
 - b) If it is a “sequence” structure node, generate a set of test cases with each test case corresponding to an ordered combination of the child nodes.
 - c) If it is a “choice” structure node, generate a set of test cases with each test case corresponding to one child node.
 - d) If it is an “all” structure node, generate a set of test cases with each test case corresponding to a random combination of the child nodes.

Data types can also be combined to form aggregate or collection data types such as *List* or *Union*. Similar to the complex data type analysis, different algorithms are developed to generate sets of test cases based on the constraint relationships among the member data types. Service providers can also define their own

service-specific data types as user-derived data types which are derived from built-in data types. A user-derived data type is first analyzed taking the knowledge from the built-in type that it is derived from. Specific derivation features are analyzed at runtime case by case.

3.2. Operation flow generation based on dependency analysis

A service may contain multiple operations. Test cases are generated for testing individual operations as well as a sequence of operations called operation flow. Based on the test data generation, multiple test cases can be generated for an operation by combining its parameter data sets. Furthermore, sets of test cases can be generated to test complex operation flows of a service by operation dependency analysis and operation combination.

The set of operations inside a service definition may depend on each other. For example, for the StockQuote service, the user can access the “GetLastTradePrice” operation if and only if it has performed “Login” operation and an authenticated identification is returned. Hence, the “GetLastTradePrice” operation is dependent on the “Login” operation. Tsai et. al discussed the extension of WSDL to facilitate test case generation[8]. Particularly, they propose four extensions to current WSDL specification: input-output dependency, invocation sequences, concurrent sequences, and hierarchical functional description. Input-output dependency identifies the association between interfaces, which can help to reduce the number of test cases for regression testing. Invocation sequences capture data flow and control flow between services while concurrent sequences capture the concurrent behavior of multi-service transactions. The sequence specifications can be used for path testing. Hierarchical functional description presents the interface functional specifications in a hierarchical structure, which can facilitate automatic functional testing.

This paper is based on current WSDL specification and analyzes the data dependences among operations. Three types of dependencies are defined: input dependency (ID), input/output dependency (IOD), and output dependency (OD). An operation *op1* is said to be input/output dependent on another operation *op2* if at least one of the input messages of *op1* is the output message of *op2*. Two operations are said to be input dependent if they both share a set of messages as their input message. Similarly, two operations are said to be output dependent if they both share a set of messages as their output message. We use *Input (op)* to represent the set of input messages of an operation *op* and

Output (*op*) as the set of output messages, the definitions are given as follows:

Definition 1 Operation Input Dependency: If $Input(op_1) \cap Input(op_2) \neq \emptyset$, then op_1 and op_2 are input dependent, that is, $ID(op_1, op_2)$.

Definition 2 Operation Output Dependency: If $Output(op_1) \cap Output(op_2) \neq \emptyset$, then op_1 and op_2 are output dependent, that is, $OD(op_1, op_2)$.

Definition 3 Operation Input/Output Dependency: If $Input(op_1) \cap Output(op_2) \neq \emptyset$, then op_1 is input/output dependent on op_2 , that is, $IOD(op_1, op_2)$. op_2 is said to be the precedent of op_1 , and op_1 is said to be the dependent of op_2 .

IOD satisfies the transitive properties. That is, for any three operations op_1 , op_2 and op_3 , if $IOD(op_1, op_2)$ and $IOD(op_2, op_3)$, then $IOD(op_1, op_3)$. *IOD* imposes sequence constraints on the operations. An operation is expected to be exercised before its dependent operations and after all its precedent operations. Therefore, a recursive *IOD* analysis process is first performed and a dependence diagram is created. The operations are arranged in order according to their *IOD* relationships.

An operation *op* may have multiple precedents recorded as $PRE(op)$. A complete test requires a full combination of operations in PRE set. When the number of operations is large, this combination can be explosive. Output dependency can be used to categorize the precedent operations into different groups. Two operations that are output dependent are divided into separate sets. The operations in different groups are not combined, and each will be further developed into different test cases. In this way, it can greatly reduce the number of test cases generated.

Similarly, an operation *op* may have multiple dependents recorded as $DEP(op)$. Input dependency can be used to categorize the dependent operations into different group. Input dependent operations are divided into separate groups corresponding to different test cases.

The following algorithm describes the overall process of dependency-based operation flow generation:

1. For each operation, perform *IOD* analysis and identify the set of its precedents PRE and the set of its dependents DEP .
2. For each operation, perform *OD* analysis on its PRE and *ID* analysis on its DEP . As a consequence, a set of sub-groups are created, that is, $PRE = \{pre_i\}$ and $DEP = \{dep_i\}$.
3. Select an operation *op* whose PRE is empty.

4. IF $DEP(op)$ is not empty, create a set of test cases as $TC[]$.
5. For each $dep_i(op)$ in $DEP(op)$, create a test case $OP[]$ as a vector of operations, and add *op* and each operation in $dep_i(op)$ to $OP[]$.
6. Repeat 4-5 until all $DEP(op)$ are traversed.
7. Repeat 3-6 until all operations with PRE empty are traversed.

There may exist isolated operations with empty PRE and DEP . In this case, since no more semantic information can be obtained from current WSDL specification, test cases are generated by randomly combining the operations based on certain coverage criteria.

3.3. Service test specification

```

<operation-name name="operation">
  <input-name name="input">
    <message-name name="message">
      <part-name name="part">part-
value</part-name>
    </message-name>
  </input-name>
  <output-name name="output">
    <message-name name="message">
      <part-name name="part">part-
value</part-name>
    </message-name>
  </output-name>
</operation-name>

```

Figure 3 STS schema definition

The generated test cases are encoded in XML called Service Test Specification (STS). STS can be easily exchanged between distributed testing components or bound to network protocols such as SOAP for test execution. STS takes the concepts of WSDL including operations, part, message, input and output, reflecting a nature mapping between WSDL elements and test elements. The schema definition is shown in Figure 3.

4. Test coverage analysis

WSDL provides the basis for function-based test cases generation and automatic service functional testing. The test case generation algorithm parses through the entire specification and generates test cases for every interface function defined in WSDL. Based on the WSDL schema, various coverage criteria can be applied to guide and evaluate generate results. This paper examines the coverage at four levels: part coverage, message coverage, operation coverage and operation flow coverage.

- A part is mapped to a parameter of an operation.

Part coverage criteria specify the rules for generating test data for each parameter. For example, each part should be covered by at least one positive and one negative test case; or each equivalent class of the part data should be covered by at least one test case.

- The input message specifies the signature of an operation, and the output message specifies the expected testing results. Message coverage criteria regulate the completeness of testing regarding to input/out domain. For example, each input message should be covered by at least one positive and one negative test case; and each equivalent class of the output message should be covered by at least one test case.
- Operation coverage specifies the extents of functions to be covered. For example, each operation (i.e. interface function) should be covered by at least one positive and one negative test case.
- Operation flow coverage specifies how execution paths need to be covered. For example, based on the dependency analysis, each path should be covered by at least one positive and one negative test case.

An experiment was conducted on 356 service WSDL specifications searched from the Internet, including address finder, airport information, etc, with altogether 2050 operations. 8200 test cases are generated covering all the operations and messages. For each operation, test cases are generated from following aspects: 1) data constraints analysis and boundary value generation; 2) random valid value within the boundary; 3) random invalid data outside the boundary; 4) operation dependency analysis; 5) operation flow generation. About 7500 test cases are successful exercised on 1800 operations. The failed test cases are mostly due to errors in WSDL files.

5. Conclusion and future work

This paper presented the process and algorithms of automatic test case generation based on WSDL specification for distributed service testing. The case study showed that this proposed approach could effectively facilitate test automation and greatly increase test productivity. However, with WSDL, only test cases for individual services could be generated, either single operations or combination of operations. In addition, with current WSDL specification, test data and operations were generated mostly by syntactical analysis such as data type analysis and operation dependency analysis. Future research will extend

current work from following perspectives: 1) Service flow specification analysis and automatic test generation for composite services; 2) Semantic service specification analysis to explore more semantic information for more intelligent test case generation.

References

- [1] J. Bloomberg, "Web Services Testing: Beyond SOAP", ZapThink LLC, Sep 2002, <http://www.zapthink.com>
- [2] B. De, "Web Services - Challenges and Solutions", WIPRO white paper, 2003, <http://www.wipro.com>.
- [3] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based verification of web service compositions", In Proc. ASE'03, 2003.
- [4] N. Looker and J. Xu, "Assessing the Dependability of SOAP RPC-Based Web Services by Fault Injection", Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03), pp. 163-163, Oct. 2003.
- [5] S. Nakajima, "Model-checking verification for reliable web service", In Proc.OOPSLA'02 Workshop on OOWeb Services, 2002, .
- [6] S. Narayanan and S. McIlraith, "Simulation, verification and automated composition of web services", In Proc. WWW'02. ACM, 2002.
- [7] H. Huang, W.T. Tsai, R. Paul, Y. Chen, "Automated Model Checking and Testing for Composite Web Services", 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Seattle, May 2005, pp. 300 - 307.
- [8] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending WSDL to Facilitate Web Services Testing", Proc. of IEEE HASE, 2002, pp. 171-172.
- [9] W. T. Tsai, Y. Chen, and R. Paul, "Specification-Based Verification and Validation of Web Services and Service-Oriented Operating Systems", 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 05), Sedona, February 2005.
- [10] X. Yi and K.J. Kochut, "A CP-nets-based Design and Verification Framework for Web Services Composition", Proceedings of the IEEE International Conference on Web Services, pp. 756-760, March, 2004.
- [11] XML Schema Part 2: Datatypes, <http://www.w3.org/TR/xmlschema-2/>, 2 May 2001.
- [12] Web Services Architecture[s]. W3C Working Draft, <http://www.w3.org/TR/ws-arch/>, Nov. 14 2002.
- [13] Web Services Description Language (WSDL 1.1) [s]. W3C Note, <http://www.w3.org/TR/WSDL/>, 15 March, 2001.
- [14] eTest, available at <http://www.empirix.com>.
- [15] eValid, available at <http://www.soft.com>.
- [16] SOAPtest, available at <http://www.parasoft.com>.
- [17] WS-I, available at <http://www.ws-i.org>.